Full Configuration Interaction Algorithm on a Massively Parallel Architecture: Direct-List Implementation

ELDA ROSSI,[1] GIAN LUIGI BENDAZZOLI,[2] STEFANO EVANGELISTI[2]

[1]*CINECA, Via Maghanelli 6 / 3, I-40033 Casalecchio di Reno (Bologna), Italy*
[2]*Dipartimento di Chimica Fisica e Inorganica, Università di Bologna, Via le Riaorgimento 4 I-40136 Bologna, Italy*

Received 4 July 1997; accepted 28 November 1997

ABSTRACT: A parallel full configuration interaction (FCI) code, implemented on a distributed memory MPP computer, has been modified in order to use a direct algorithm to compute the lists of mono- and biexcitations each time they are needed. We were able to perform FCI calculations on the ground state of the acetylene molecule with two different basis sets, corresponding to more than 2.5 and 5 billion Slater determinants, respectively. The calculations were performed on a Cray-T3D and a Cray-T3E, both machines having 128 processors. Performance and comparison between the two computers are reported and discussed. © 1998 John Wiley & Sons, Inc. J Comput Chem 19: 658–672, 1998

Keywords: full configuration interaction; *ab initio* methods; acetylene molecule; parallel computation; message passing

Introduction

Configuration interaction (CI) is a way to improve the description of a molecule by permitting the superposition of many configurations corresponding to different orbital occupancies. If no selection of the possible configurations is performed, the approach is called full-CI.[1–11] For a number of years, our group has been working on the design of a full-CI-type algorithm (FCI) on massively parallel architectures (MPP), in particular the Cray MPP machines Cray T3D and Cray T3E.[12–15] Both machines are presently available in CINECA Computer Centre in a 128-processor configuration. In this article, we present the last results in the parallel code design.

Our main goal was to introduce ''direct list'' computation in the program.[16] With this feature we intended to simplify both the structure of the

Correspondence to: E. Rossi

Contract/grant sponsors: Italian Ministry of Scientific Research (MURST), Italian National Research Council (CNR), and the European Economic Community; contract/grant number: ERBFMRXCT96/0088

CCC 0192-8651 / 98 / 060658-15

package and the data organization in central memory. On the other hand, ''direct list'' computing required a modification in the algorithm structure that would have introduced a relevant increase in data communication with the original data distribution. We were able to solve this problem by changing data distribution among the processor local memories.

At the same time, we had the opportunity to use the Cray T3E machine. We ported the program to the new computer with nearly no trouble, obtaining a reasonable performance ratio. Thanks to the larger memory available on the Cray T3E and to the data rationalization from direct list computing, we could afford to treat larger systems. In particular we studied the acetylene molecule (with 10 active electrons) by describing it with two different atomic basis sets (30 and 32 orbitals). The full-CI resulted in a 2.5 and 5 billion determinant space, respectively, provided that the D_{2h} space symmetry and a partial spin symmetry are taken into account.

In treating such large problems, the I/O activity increases tremendously. For example, in the case of the 5 billion determinant problem, the needed disk storage was in excess of 40 gigabytes with a total data movement of more than 250 gigabytes per iteration. Therefore, to obtain the full-CI result we adopted a two step strategy. At the beginning of the iterative procedure, we introduce a threshold value to compress CI vectors when written to disk, reducing disk requests and time for I/O activity. In this way we compute a first approximation to the CI eigenvector. In the second step, to ensure obtaining the exact full-CI result, we run the last part of the iterative process, until the convergence, with a strictly zero threshold. Finally, the performance of the optimized full-CI code is discussed and compared to that of the previous version, both for a nine million determinant CI space benchmark and for the actual acetylene system.

''Direct List'' Approach

The original MPP program, derived from a nonparallel version of the code (Unix workstation and Cray PVP systems) was the last stage in a more complex (three-stage) package.

1. The first program, running on a conventional machine, was intended to precompute all necessary lists, namely, monoexcitation ($\mathscr{L}^{*}_{*}$) and biexcitation ($\mathscr{L}^{\ \ *\ *}_{*\ *}$) lists, and to store them on suitable disk files.
2. The second program, running on Cray MPP machine, adapted these lists to a given T3D configuration, in terms of a fixed number of processors.
3. Finally, the actual FCI program performed the iterative procedure to get full-CI results.

To simplify this organization and to avoid unnecessary data structures (precomputed list items require in some cases a large amount of memory space), an algorithm change was introduced. In the present version of the code we do not read list information from file but we compute it ''on demand,'' when needed. We call this approach the ''direct list'' algorithm (refer to Gagliardi et al.[16] for notation).

The full-CI algorithm is roughly composed by three main routines,[14,17] alpha-beta, beta-beta, and Y-transpose routines, called in sequence. The first two routines require about 90% of the CPU computing time. Two types of lists are needed.

- Monoexcitation ($\mathscr{L}^{*}_{*}$) lists are required at two different points of the alpha-beta routine. For each given pair (i, j) of orbitals, there is a list $\mathscr{L}^{j}_{i}$ of all the strings having i-orbital occupied and j-orbital virtual (empty) and a list $\mathscr{L}^{i}_{j}$ of the corresponding configurations obtained by moving the electron from the i-orbital to the j-orbital.
- Biexcitation ($\mathscr{L}^{\ \ *\ *}_{*\ *}$) lists are used in the beta-beta routine. For each given set (i, j, k, l) of orbitals, there is a list $\mathscr{L}^{jl}_{ik}$ of all the strings having i and k orbitals occupied and j and l orbitals virtual and a corresponding list $\mathscr{L}^{ik}_{jl}$ of the configurations obtained by moving two electrons from the i and k orbital to the j and l orbitals.

The dimensions of the lists depend on the number of orbitals and electrons. This becomes very important and, to some extent, hard to predict, in large systems such as those with which we are dealing here. As a preliminary step, we create an ordered list of all the strings of our system, i.e., all the electronic configurations one can obtain distributing all the electrons in the available orbitals.

In the example shown in Figure 1, the ordered list of the 20 strings is shown, relative to the distribution of three electrons into six spin orbitals.

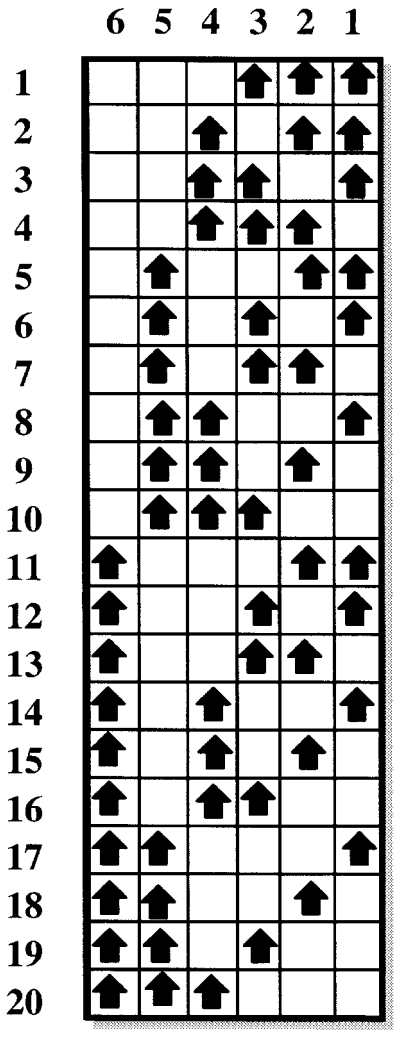

FIGURE 1. Ordered list of configuration strings for a system with six orbitals and three electrons.

This list will be searched sequentially each time that is needed. For example, in the alpha-beta routine, the $\mathscr{L}^*_*$ lists will be computed in the following way. Given a pair of orbitals, i and j, the configuration list will be searched for

- all the ''starting'' strings, i.e., all the configurations having an electron in the orbital i and a hole in the orbital j ($\mathscr{L}^j_i$), and
- all the ''destination'' strings, i.e., all the configurations obtained from the original ones by moving the electron from i to the orbital j ($\mathscr{L}^i_j$).

From a practical point of view, the two lists are obtained by independent searchings in the ordered list, the correspondence being ensured by the string generation order. A detailed description of the list generation can be found in Gagliardi et al.[16]

Because the ''direct-list'' routine is called an enormous number of times (inside a four-way loop running on the orbitals), it was important to optimize it. For this reason the routine was designed with orbitals indices as input data and the two lists of ''starting'' and ''destination'' strings as output data, even if this forced us to modify the algorithm substantially.

Data Distribution Modification

In introducing the direct-list computing into the two basic alpha-beta and beta-beta routines, we realized that we had to change the order of some loops. This operation, which is trivial in a conventional environment, becomes hard work in a parallel message-passing environment where data sharing is used to distribute data and work. In our case, in particular, the new loop order introduces an unbearable overhead in data communication, which we have worked out by changing the data topology.

As is usual in full-CI algorithms, transformed integrals do not represent a problem from the point of view of storage requirements; they grow as the fourth power of the basis set size, whereas the coefficient vector has combinatorial behavior.[16] For example, in the largest case we report in this paper, the integrals are only about 20,000, compared to the 16 million words available in the memory of each node. A replicated data solution has been adopted to store the integral data structure, because this is the simplest solution when central memory saving is not required.

The main data structures of the algorithm consist of two arrays (X and Y), containing the coefficients of the CI wave function and the iterative correction to the wave function itself.[12,16] In Figure 2 private data structures (i.e., stored in the memory of each of the involved processors) required by the two versions of full-CI code are reported. In the original code, the arrays X and Y are shared so that each local memory holds a consecutive set of columns. Two additional work areas are needed in each local memory, namely, Xv, dimensioned as a column of X, and Ya, dimensioned as Y. The memory request for Ya can be reduced, if not enough memory is available, using a buffering technique (the number of columns remains un-

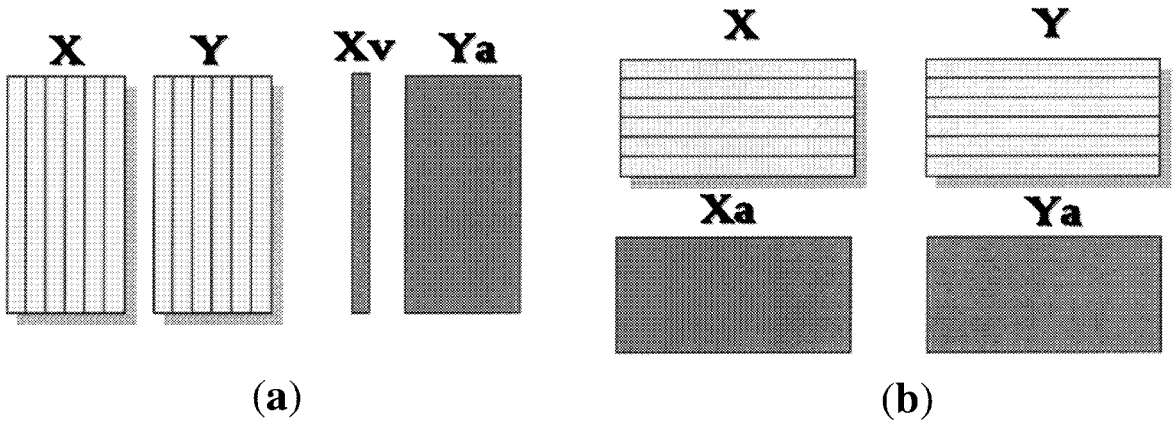

FIGURE 2. Single-processor data structures in the two versions of full-CI. Coefficient arrays (in light grey) are distributed by columns in the original version (a) and by rows in the new version (b). Moreover, work area structures (in dark grey) differ in shape and dimension.

changed and the number of rows can be reduced according to memory availability).

In the modified version of the code, the arrays X and Y are shared so that each processor's local memory holds a consecutive set of rows. Two specific work areas are needed in each local memory, Xa, dimensioned as X, and Ya, dimensioned as Y. Again, the memory request for Xa and Ya can be reduced, if not enough memory is available, accordingly reducing the number of rows. This new data topology is characterized by some drawbacks, in particular an increased dimension work area is requested, and a theoretical reduction in the performance of the algorithm is possible.

In fact, from a computational point of view, the most important part of the algorithm is a SAXPY operation, i.e., Vector_1 = Vector_1 + scalar * Vector_2, whose performance depends on the length of the working arrays. In the full-CI algorithm, SAXPY operations are performed between columns of X and Y matrices, which decrease their length in the new data topology. For this reason the new version of the algorithm tends to decrease its efficiency, at least when small cases are treated with a high number of processors, and data structures are not large enough (see below under The Problem of ''Data Streams'' for other comments about SAXPY performance and comparison between the two code versions).

The Beta-Beta Routine: Biexcitation Lists

In the beta-beta routine, the application of the direct list computing is straightforward. Because the list routine is best used giving the four orbital indices, an algorithm change must be made. The original beta-beta algorithm is given below.

```
Loop Xcol (β strings)
   Xv = X( :, Xcol)    (Communication!)
   access 𝒮Xcol that returns {[col- loc Y] ↔ [int]}
   loop (Ycol, int) in ([col- loc Y],"[int])
     Y( :, Ycol) = Y( :, Ycol) + int * Xv    (SAXPY)
   End loop (Ycol, int)
End loop Xcol
```

The biexcitation lists ($\mathscr{L}_{\texttt{Xcol}}$) are accessed in line 3, where for each ''starting'' string (`Xcol`) all the local ''destination'' strings (`[col- loc Y]`) are returned, together with the correct coefficients [`int`]. For data locality reasons, the main loop runs on β strings, i.e., on columns of the X coefficient ma-

trix. Every single column (which can be local or remote) is copied in the local vector Xv and then used to upgrade all the local columns of Y using a suitable coefficient. The sequence of the Y columns and the corresponding coefficients for each given column of X are retrieved from the list item $\mathscr{L}_{\texttt{Xcol}}$.

Because direct list routines require the four orbital indices as input, we have to change the main loop into a four-way loop running on orbital indices. Because the data topology is changed too, the beta-beta routine becomes completely local, as can be seen in the following scheme.

```
Loop i, j, k, l (β orbitals)
    access 𝓛ijkl, 𝓛klij that return {[col-tot Y] ↔ [col-tot X]}
    loop (Ycol, Xcol) in ([col-tot Y], [col-tot X])
      Y(:, Ycol) = Y(:, Ycol) + int(i, j, k, l) * X(:, Xcol)    (SAXPY)
    End loop (Ycol, Xcol)
End loop l, k, j, i
```

The main operation is a SAXPY operation that works on contiguous elements arranged into local columns of X and Y matrices. The length of these columns is roughly given by `N/npe` (the total number of strings divided by the number of processors), because of the data topology (see Fig. 2). This can in principle reduce the performance of the SAXPY operation. However, we verified that on the Cray T3E the SAXPY performance is saturated quickly with respect to the vector length (see below under The Problem of ''Data Streams.''

A more important drawback is due to the fact that each processor must compute the complete list sequences in exactly the same way. The time necessary for list computing becomes an important fraction of the total time, particularly for a large number of processors, because the list computation does not scale with the processor number. We tried a distributed computation (work sharing) approach, where each processor computes only some of the lists and shares them with the others, but this introduced a severe synchronization problem. Probably a mixed approach is the best solution, provided a certain quantity of memory is available. However, we did not explore this point further.

The Alpha-Beta Routine: Monoexcitation Lists

Modifying the alpha-beta routine was more troublesome, owing to the more complex structure of the algorithm. An outline of the original alpha-beta code is given in the following scheme.

```
Loop i, j (α orbitals)
    access 𝓛ij, 𝓛ji(α) that return {[row-tot Y] ↔ [row-tot X]}
    Ya = 0
    Loop Xcol (β strings)
      Xv = X([row-tot X], Xcol) (Gather + Communication!)          (a)
      access 𝓛Xcol(β) that returns {[col-loc Y] ↔ [int]}
      Loop (Ycol, int) (in [col-loc Y], [int])
        Ya(:, Ycol) = Ya(:, Ycol) + int*Xv    (SAXPY)            (b)
      End loop (Ycol, int)
    End loop Xcol
    Scatter Ya in Y (using [row-tot Y] from 𝓛ij(α))               (c)
End loop j, i
```

The two outer loops (on i and j) run on α orbitals. For each pair of orbitals a list item is accessed, stating the relationship between X and Y rows (in the global range, because each processor holds all the matrix rows in its local memory). Then, the inner loop (on β strings) is entered. Because the operation to be performed here is a combination (SAXPY) between columns not always local, we minimize data communication by getting one X column at a time and using it for updating every possible local column of Y. For this reason, the inner loop runs over strings (column of X). When the selected column is not local, a remote (get plus gather) operation is performed, using `[row-tot X]` data for the element selection inside the column. The selected gathered column is stored in Xv (a), and a list item is accessed for it returning all the Y local columns that should be modified and the corresponding integral to be used. The selected X column (in Xv) is then used to upgrade all the corresponding Y columns (b), using a temporary storage array Ya. Only when all β strings have been taken into account are the Ya rows scattered locally into Y (c). It is important to note that the monoexcitation list generation is required twice, for the α electrons and for the β electrons. Even if the two instances are equivalent in principle, in practice we must access two different kind of information because of the way the α and β loops are organized.

To insert the direct-list routines in this algorithm, we had to change the inner loop, which becomes a double loop on orbitals similar to the beta-beta routine. The new data topology, distributing the X and Y arrays by groups of contiguous rows, aims to maintain locality inside the inner loops, resulting in low data communication. The alpha-beta routine, with direct-list features, is shown below.

```
Loop i, j (α orbitals)
    access 𝒮_i^j, 𝒮_j^i(α) that return {[row-loc Y] ↔ [row-tot X]}
    Gather X in Xa (using [row-tot X]) (Communication!)                (a)
    Ya = 0
    Loop k, l (β strings)
      access 𝒮_k^l, 𝒮_l^k(β) that return {[col-tot Y] ↔ [col-tot X]}
      loop (Yrow, Xrow) (in [col-tot Y], [col-tot X])
        Ya(:, Ycol]) = ... + int*Xa(:, Xcol) (SAXPY!)                  (b)
      End loop (Yrow, Xrow)
    End loop l, k
    Scatter Ya in Y (using [row-loc Y] from 𝒮_ij(α))                   (c)
End loop j, i
```

The $\mathscr{S}^*_*(\alpha)$ list routine now returns slightly different information and becomes ''local,'' because the algorithm requires only those rows of X that are needed to update local rows of Y. The selected X rows (both local and remote) are then collected in the Xa work area (a). The inner loop changes to allow direct list invocation, and the β strings are processed in terms of pairs of orbital indices (k, l). For each such pair, a $\mathscr{S}^*_*(\beta)$ list is obtained to establish correlation between X and Y columns. The list is used for the main SAXPY operation (b) when Y columns are updated using corresponding X columns. This is done preliminary with Xa and Ya work areas. Only when all β strings have been considered, are Ya rows scattered into corresponding Y rows using the $\mathscr{S}^*_*(\alpha)$ list (c). Data communication, in this new version, is limited to the ''row gather'' operation; from that point on, all the processors can work on local data, and no synchronization is needed at all.

Regarding the beta-beta routine, one drawback of this organization is that the $\mathscr{S}^*_*(\beta)$ determination becomes ''global''; given four orbital indices (i, j, k, l), each processor must compute exactly the same piece of information, and no work sharing is possible. The problem is particularly severe because the $\mathscr{S}^*_*(\beta)$ list determination is done inside the inner loop. Another disadvantage is due to the fact that the basic SAXPY operation (b) tends to be carried out on shorter vectors and for this reason

has lower performance. As was discussed for the beta-beta case, these facts do not represent a major problem in the algorithm.

Using a Threshold Value to Compress CI Vectors

The algorithm requests two coefficient arrays, X and Y. The X array, containing the CI coefficients, must be read eight times (in D_{2h} symmetry) for each iteration, in order to apply the Hamiltonian to X and to compute the correction Y array. Moreover, both arrays X and Y are read and written once for each iteration. This results in 12 read and write operations, with a total data movement that, in the largest case we run (acetylene, with 32 orbitals), is in excess of 250 gigabytes per iteration. If we consider the typical throughput values for disks available on Cray computers (SCSI: 10 Mbytes/sec; DD60: 20 Mbytes/sec) we realize both the huge amount of time spent in I/O activities and the importance of new strategies to reduce this time. For this reason we implemented a data-compression routine that reduces both I/O activity and disk occupation, cutting from X and Y matrices all the elements smaller than a given threshold value (typically 10^{-5}–10^{-8}). This compression technique is used for the computation of a very accurate guess at or first approximation to the CI eigenvector. Starting from this guess, usually a small number of further iterations with no compression is sufficient to achieve convergence. We note that other compression techniques[18] have been used for large full-CI calculations.[19]

Let us consider an array V whose elements in modulus are all strictly lower than 1.0 (if this is not the case, each element must be divided by a suitable constant factor). The array V can be compressed in the following way, cutting all elements smaller than a given threshold.

```
ipos = 0
Loop i (length of V)
   if (ABS(V(i))>threshold) then
     counter = counter + 1
     V(counter) = ABS(V(i)) + i, with the sign of V(i)
   end if
End loop i
tot_length = counter
```

At the end of the routine, the first `tot_length` positions of V will contain the elements of the original array V that are (in modulus) greater than the given threshold. Each one is summed with the position i in the original array, and the sign is given by the sign of the original element.

In this way we can use the vector V itself to store the compressed array. Using 64-bit words in a IEEE standard format, it is possible to compress a vector of about 10 million words (this is an upper limit to the dimension of the local vector with which we are working) while keeping seven or eight meaningful figures. We verified that this level of numerical precision is largely sufficient to obtain a highly accurate guess. This compression technique is very effective in reducing the amount of data to be transferred to and from the discs, in particular during the beginning of the iterative process, where the coefficient arrays are not very well populated. For example, in the case of acetylene described in the next section, after 10 iterations, only 15% average of the coefficients in the X matrix were larger than the threshold used, 10^{-8}. The disc space needed to store the X matrix was about 1.5 gigabytes instead of the 10 gigabytes necessary for the uncompressed matrix.

To reexpand the compressed array V, we must proceed backwards, to avoid overwriting the part of V already uncompressed.

```
V(tot_length+1 : )=0.0
Loop i (backward from tot_length to 1)
   position=INT(ABS(V(i)))
   V(position)=ABS(V(i))-position, with the sign of V(i)
   if (position < > i) V(i)=0.0
End loop i
```

In this way, we can reduce the I/O time significantly, and the time requested for compressing and uncompressing data is practically negligible. Moreover, the guess computation, although rather time consuming, requires a very small disk allocation. Finally, we note that this technique can be used to store efficiently, in an approximate way, the full-CI vector resulting from a large calculation.

Ground State of C_2H_2

We have used our parallel code to investigate the full-CI energy of the ground state of acetylene, C_2H_2. Two different basis sets have been used, hereafter indicated as B1 and B2, differing for the number of functions on the hydrogen atoms: B1 refers to a *[3s2p1d / 2s]* basis set and B2 to a larger *[3s2p1d / 3s]* basis set.

The two *1s* orbitals of the carbon atoms have been kept doubly occupied in all the calculations and frozen to the optimized form obtained through a valence CAS-SCF calculation. This implies that the size of the two basis sets is N1 = 30 and N2 = 32 active orbitals. Because we have a total of 10 active electrons, this makes two full-CI spaces of 2,538,565,366 and 5,069,065,864 determinants in D_{2h} symmetry. They are close to the limiting size we are able to treat with our code on CINECA T3D and T3E, respectively. A single point near the equilibrium geometry has been computed with the larger basis set, B2, and the total symmetric stretching of the C—C and C—H bonds has been investigated with the smaller basis set, B1.

Two vector symmetry blocks are needed in memory at the same time, a block of X and a block of Y. As was explained in the previous section, the compression technique was used to compute a very accurate guess for the exact wave function. A threshold of 10^{-8} was chosen for cutting the coefficients during the guess computation.

Timing for the B1 basis set acetylene is reported in Table I. We run the complete calculation on 96 T3E processors (the minimum number to fit the problem without buffering). For comparison purposes only, one iteration was also executed on 128 processors, both on T3D and T3E. Two different kinds of iterations are reported: the first used to compute the guess, with reduced I/O activity (threshold of 10^{-8}), and one of those used for real full-CI convergence, with complete I/O.

Timing for the B2 basis set acetylene is reported in Table II. We run the complete calculation on the 128-processor T3E, reducing the total work area with a buffering technique. Both kind of iterations, with reduced and full I/O, are reported.

The analysis of the full-CI vector for the B2 acetylene at the equilibrium geometry is reported in Table IIIa. The population of the vector is reported as a function of the size of the coefficients. For each class, we report the total number of coefficients (`count`), the sum of their absolute values ($\sum|x|$), and the sum of their squares ($\sum|x^2|$). The most populated class is the (10^{-9}–10^{-10}) one, whereas the sum of the absolute values reaches its maximum for the (10^{-4}–10^{-5}) class. The vector is very much concentrated in the large-threshold classes. This explains why the introduction of even

TABLE I.
Timing Summary (in Seconds) for B1 Acetylene on Different Configurations of MPP Computers: Computation of the Guess (Reduced I / O, Threshold = 10^{-8}) and of the Complete Iteration (Full I / O).

	T3D (128 PE) reduced I / O	T3E (128 PE) reduced I / O	T3E (96 PE) reduced I / O	T3E (96 PE) full I / O
Alpha-beta	12,709	3,649	4,422	
Beta-beta	1,650	635	763	
I / O	85	45	53	12,617
Total (one iteration)	14,520	4,375	5,286	17.850

TABLE II.
Timing Summary (in Seconds) for the B2 Acetylene on a Cray T3E Computer: Computation of the Guess (Reduced I / O, Threshold = 10^{-8}) and of the Complete Iteration (Full I / O).

	T3E (128 PE) reduced I / O	T3E (128 PE) full I / O
Alpha-beta	10,250	
Beta-beta	1,664	
I / O	1,053	20,915
Total (one iteration)	12,770	32,903

a very small threshold results in major disk savings and permits us to obtain very accurate guesses of the full-CI vector.

It would be interesting to evaluate the effect of decreasing thresholds on the convergence of the energy towards the full-CI limit. This can be done by iterating until energy stabilization for each decreasing threshold value, from 10^{-1} to 0. Because the B2 acetylene case is too heavy for such a systematic investigation, we performed this analysis for a more affordable case, the nitrogen molecule (N_2), with five active electrons and 18 active orbitals. In Table IIIb the result is reported,

TABLE IIIa.
Analysis of the Full-CI Vector: B2 Acetylene, Equilibrium Geometry.

Thresholds	Count	$\Sigma \lvert x \rvert$	$\Sigma \lvert x^2 \rvert$
$1-10^{-1}$	3	1.17347732	0.88691345
$10^{-1}-10^{-2}$	184	3.25859248	0.07526284
$10^{-2}-10^{-3}$	3,094	8.20274348	0.03222839
$10^{-3}-10^{-4}$	67,071	14.48635690	0.00461254
$10^{-4}-10^{-5}$	894,068	22.81574050	0.00088611
$10^{-5}-10^{-6}$	8,831,527	22.67406962	0.00008869
$10^{-6}-10^{-7}$	63,165,211	17.79195074	0.00000757
$10^{-7}-10^{-8}$	290,400,449	8.80592742	0.00000040
$10^{-8}-10^{-9}$	865,918,188	2.88300373	0.00000001
$10^{-9}-10^{-10}$	1,493,284,047	0.55440300	0.00000000
$10^{-10}-10^{-11}$	1,415,352,464	0.05890022	0.00000000
$10^{-11}-10^{-12}$	694,274,827	0.00323100	0.00000000
$10^{-12}-10^{-13}$	188,322,088	0.00009369	0.00000000
$10^{-13}-10^{-14}$	40,172,592	0.00000201	0.00000000
$10^{-14}-0$	8,380,051	0.00000004	0.00000000
Total	5,069,065,864	102.70849219	1.00000000

TABLE IIIb.
Analysis of the Full-CI Vector and Energy Contributions for Different Threshold Values: Nitrogen Molecule.

Thresholds	Count	$\Sigma \lvert x \rvert$	$\Sigma \lvert x^2 \rvert$	Energy contrib.
$1-10^{-1}$	3	1.18277596	0.91538778	−108.96718722
$10^{-1}-10^{-2}$	158	2.81331113	0.06897745	−0.18792832
$10^{-2}-10^{-3}$	1,177	3.18145930	0.01377007	−0.07543627
$10^{-3}-10^{-4}$	19,907	4.75854035	0.00169563	−0.01038065
$10^{-4}-10^{-5}$	125,001	3.60787335	0.00015961	−0.00173990
$10^{-5}-10^{-6}$	712,402	2.08615642	0.00000912	−0.00012203
$10^{-6}-10^{-7}$	1,988,206	0.67642300	0.00000034	−0.00000648
$10^{-7}-10^{-8}$	3,169,824	0.12061179	0.00000001	−0.00000016
$10^{-8}-10^{-9}$	2,268,949	0.01011318	0.00000000	−0.00000001
$10^{-9}-10^{-10}$	699,777	0.00035390	0.00000000	0.00000000
$10^{-10}-10^{-11}$	101,284	0.00000551	0.00000000	0.00000000
$10^{-11}-10^{-12}$	10,572	0.00000006	0.00000000	0.00000000
$10^{-12}-10^{-13}$	1,078	0.00000000	0.00000000	0.00000000
$10^{-13}-10^{-14}$	140	0.00000000	0.00000000	0.00000000
$10^{-14}-0$	78,386	0.00000000	0.00000000	0.00000000
Total	9,176,864	18.43762396	1.00000000	−109.24280104

together with the analysis of the final full-CI vector.

In this rather small case, a threshold of 10^{-8} is sufficient to stabilize the energy within a μhartree from the full-CI value. The behavior for larger systems is certainly less favorable. However, stabilization of the energy with such a threshold, followed by a few iterations with zero threshold, proved to be an efficient strategy in most cases to achieve convergence.

From Cray T3D to Cray T3E: Some Conversion Issues and Performance Considerations

Converting a program from T3D to T3E has proved to be straightforward.[20] The main topic of concern is that on a Cray T3E only the Fortran 90 compiler is supported, but the original code was written in Fortran 77. Because Fortran 77 is a subset of Fortran 90, practically no change was needed to use the new compiler. A more important issue is the lack of the specific Cray ''data parallel'' environment (called CRAFT), but this was not a concern because FCI code is entirely written with the Cray ''message passing'' library called SHMEM (shared memory) library.

This technology for message passing implementation was originally chosen for performance reasons. As far as portability considerations are concerned, we would stress that this is not our main concern; the parallel version we are describing was derived from a ''nonparallel'' version that is easily portable on all Unix-like platforms. The parallel algorithm is designed on the ''one-sided communication'' characteristic[21] of the SHMEM routines, so the translation to other libraries, such as PVM, that do not have such a feature is rather difficult. On the other end, the communication routines are used only in a few places of the code, so the translation to a library with the ''one-sided'' characteristic (MPI 2, for example) is expected to be straightforward.

To discuss performance topics and to compare different code versions and different MPP platforms, we consider a rather small molecular system, i.e., the nitrogen molecule (N_2), with five active electrons and 18 active orbitals, resulting in a 9-million-determinant full-CI space. This case is too small to be completely meaningful, in particular for the actual version of FCI code, which does not behave very well if too many processors are used. Better performance is obtained for the minimum number of processors that fit the problem dimension. Beyond that number, performance decreases rapidly because of

- shortening of the data structure interested in SAXPY operation;
- direct list time remaining unchanged in the beta-beta routine and in the inner loop of the alpha-beta routine;
- more data communication activity.

Nevertheless, the importance of this benchmark is in the fact that it is sufficiently small to be run on any number of processors, starting from two. All timing data reported from here on are relative to the N_2 case.

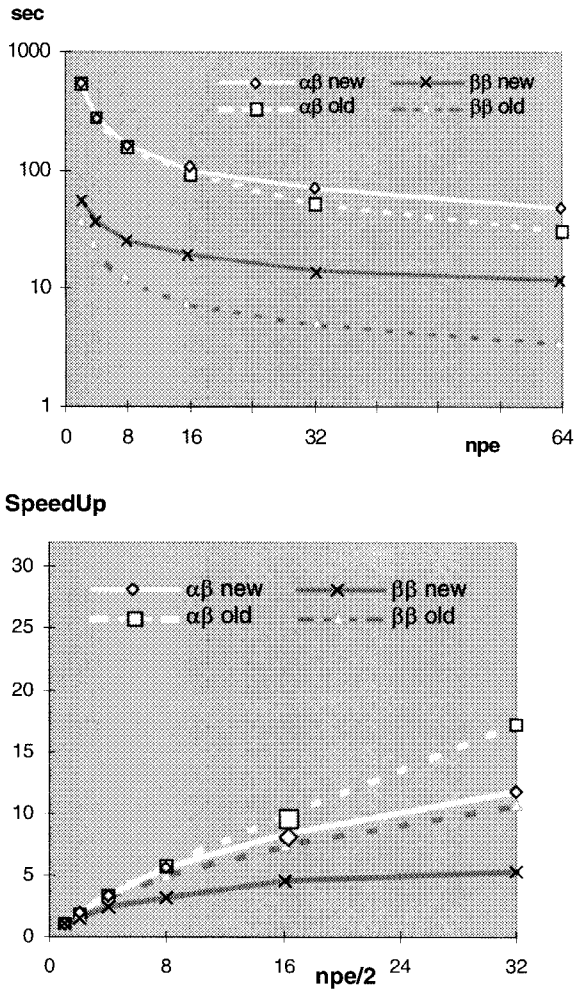

FIGURE 3. Timing comparison (Cray T3D) between the old and new versions of the algorithm for the N_2 benchmark. Seconds vs. number of processors and speed-up.

First, we compare the original version of FCI, where list items are obtained from a precomputed file, to the new version, where lists are computed on demand. In Figure 3 some timings are reported, relative to the N_2 benchmark. The first panel reports elapsed time (in seconds) vs. the number of processors; in the second panel the speed-up figure is sketched, taking the two processor case as the starting point. The original version of the program (dashed lines) is compared to that based on direct-list computing (solid lines). In particular, the two main routines are considered, namely, alpha-beta (dark lines) and beta-beta (light lines) routines. Greater differences are found for high numbers of processors, but this depends on the little dimension of the molecular system. For example, the alpha-beta routine accounts for 30 vs. 44 seconds with 64 processors, but only 514 vs. 523 seconds when two processors are used. Even if a limited loss of performance has to be accepted, we expect that it tends to decrease for the large systems in which we are interested. Moreover, we believe that this is a reasonable fee to pay compared to the advantages that can occur.

THE PROBLEM OF "DATA STREAMS"

Data streaming (or stream buffers) is a hardware feature available, at present only on request, on the Cray T3E.[22] This is able to speed up, in a substantial way, the kinds of operations that need a continuous feed of contiguous elements from central memory. At present, the Cray T3E has some problems in ensuring data coherence when using data streaming together with remote communication, so data streaming has to be enabled explicitly by the programmer.

In the new version of the FCI code, data streaming has been demonstrated to be very useful, both for alpha-beta routine (from 237 to 157 seconds with two processors) and for beta-beta routine (from 50 to 28 seconds). As is shown in Figure 4, greater speed is obtained for the SAXPY operation in both routines and in the "scatter" part of the alpha-beta routine. To avoid data coherence problems, data streaming has been enabled only within the two main routines.

As far as SAXPY behavior is concerned, we expect a reduction in performance when the new data topology is applied, because vectors involved

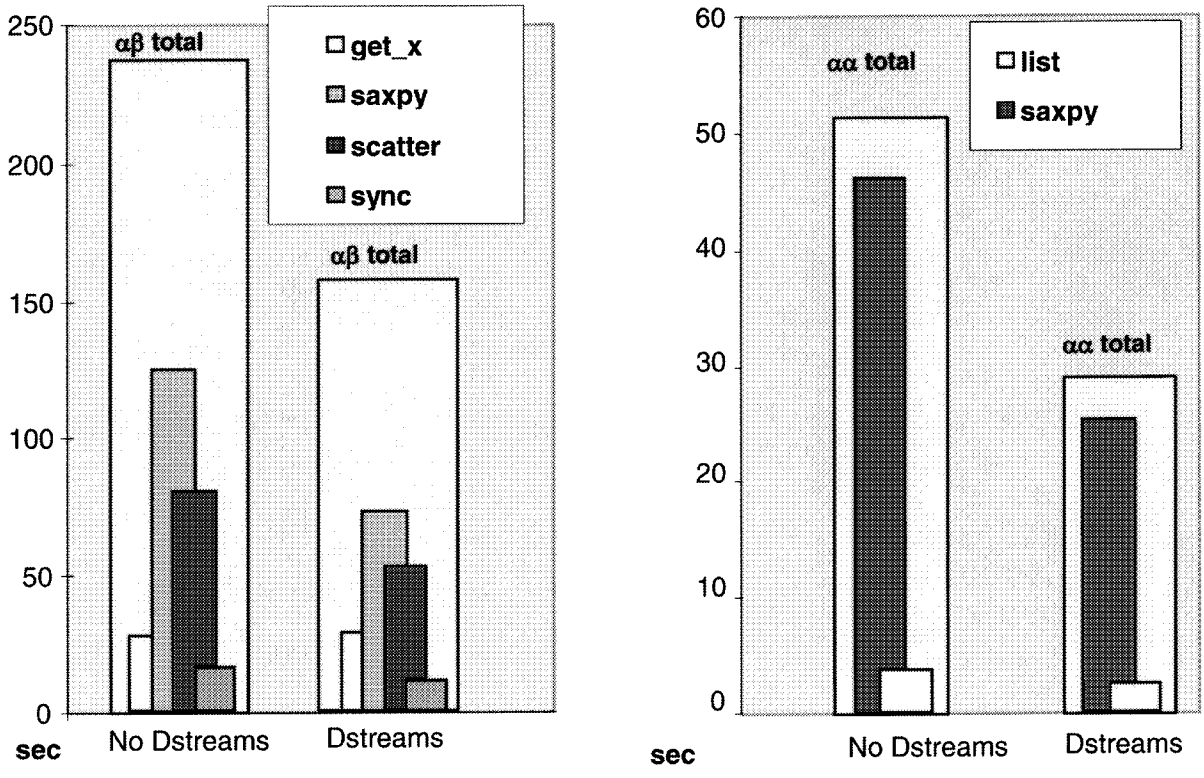

FIGURE 4. Timing comparison when using D-streams on Cray T3E; N_2 molecule, two processors.

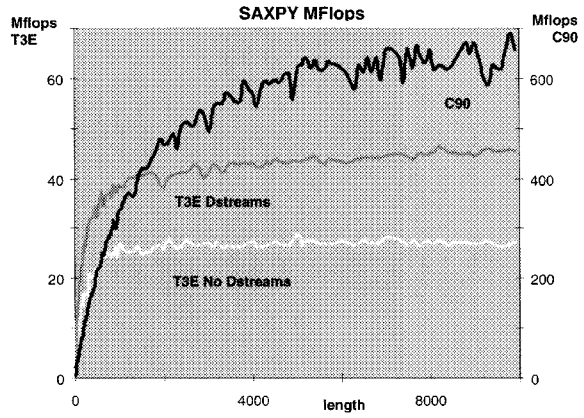

FIGURE 5. Performance (in MFlops) of the SAXPY operation on T3E and on C90, as a function of the vector length. For T3E data, performances with ''data streams'' enabled and disabled are reported. C90 performances are scaled with the right axes.

in SAXPY decrease their length with the number of processors. This fact reduces in principle the performance of the SAXPY operation. However, we have verified that on the Cray T3E SAXPY performance saturates quickly with respect to the vector length. In Figure 5 we compare SAXPY performance (in MFlops) on the Cray T3E (with and without data streams enabled) and on the Cray C90 (it is worth noting that the C90 trend is related to a different scale, reported on the right, because C90 performance can reach 700 MFlops for large vector processing). Cray T3D behavior is not indicated, but the figure is essentially the same as that obtained with the T3E when data streams are not used.

THE PROBLEM OF MOVING A STRIDDEN VECTOR IN MEMORY

When we analyzed the performance ratio between the Cray T3D and Cray T3E (see also the following section), we found an anomalous behavior in the Get part of the alpha-beta routine. The ratio between the time spent in this part of the code when running on the two different machines was about 1.5 and did not depend on the number of processors. The Get operation is responsible for data movement and communication activity. In particular, it gathers a number of rows of the X matrix into consecutive rows of the Xa work matrix. Because the source rows can be either remote or local, we have correspondingly either a vector move or a data transfer. Both of them are stridden; i.e., they refer to elements not consecutive in memory, because the Fortran compiler stores matrices by columns. In particular, for local data movement a vector syntax is used, whereas for data communication the SHMEM_IGET routine (from the Cray SHMEM library[23]) is used, as sketched in the following scheme.

```
i = 0
Loop Xrow (in [row-tot X] from 𝒮*(α))
   i = i + 1
   If (Xrow is local) then
     Xa(i, : ) ← X(Xrow, : )
   If (Xrow is remote) then
     Call SHMEM_IGET for Xa(i, : ) ← X(Xrow, : )
End loop Xrow
```

Surprisingly, we verified that the reason for the anomalous behavior is the local move activity, not the remote communication. In fact, moving in memory 1,000 vector elements with stride equal to 1,000 results in a throughput of 15 Mbytes/sec, to be compared with a 65 Mbytes/sec throughput when transferring the same vector from a different processor's memory. Moreover, we found that the best way to move a vector in memory is to use the SHMEM routine, as if it were a remote move. The results of this investigation is reported in Table IV.

TABLE IV.
Throughput Values (in Mbytes / sec) for Vector Moves (1,000 Elements) on the T3E, Both Local and Remote, With Different Stride Values and Different Techniques.

Mbytes / sec	V_1 = V_2	IGET (local)	IGET (remote)
Stride = 1	190	276	255
Stride = 1,000	15	43	63
Stride = 999	15	130	130

Stridden SHMEM_IGET operation seems to be constrained by memory bandwidth. Because the memory has interleaved banks,[22] strides greater than 1, particularly even-numbered strides, yield lower bandwidth than unit strides or odd-numbered strides. Also, insofar as memory is the bottleneck, local-to-local copies are slowed, because memory activity is occurring concurrently for the source and target arrays on the local memory. SHMEM routines make use of a specific hardware feature (E registers) that allows better performance. It seems evident that the Fortran 90 compiler cannot generate optimal code. In the next release of the Fortran 90 compiler (Programming Environment 3.0[22]), a specific compiler directive should be present to address this performance problem.

We modified the code, in order always to use the SHMEM routine for both local and remote operations. This simple step allows us to speed up the alpha-beta routine by 100 seconds (from 267 to 157) for the N_2 benchmark running on two processors. The performance ratio between the T3D and the T3E is now about 6–7, a very good value compared to the other parts of the code (see the following section).

COMPARISON BETWEEN THE CRAY T3D AND THE CRAY T3E

In Figure 6 the timing comparison between the T3D and the T3E, for the N_2 benchmark, is reported. The code is compiled with the Fortran 90 compiler Cray CF90[24] and is strictly the same on both machines, with only two exceptions:

- on the Cray T3E data streaming is enabled in alpha-beta and beta-beta routines, whereas on the T3D this is not applicable;

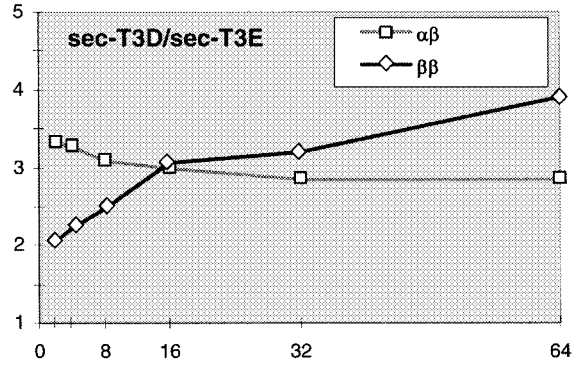

FIGURE 6. Timing comparison between Cray T3D and Cray T3E, on N_2 benchmark, for the two main routines.

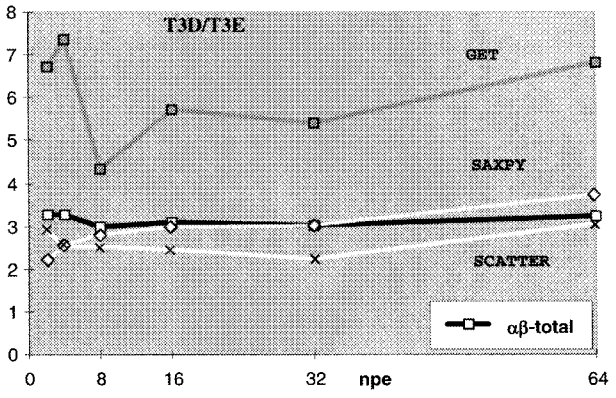

FIGURE 7. Timing comparison between Cray T3D and Cray T3E, on N_2 case, alpha-beta routine and its components.

- on the Cray T3E, vector transfers are performed with SHMEM routines regardless of locality, whereas on the T3D, only remote transfers are performed with SHMEM routines.

The theoretical performance ratio between Cray T3E and T3D is about 4, as can be obtained from reported peak performance tests of the corresponding single processors. This means 150 MFlops for the T3D vs. 600 MFlops for the T3E. Nevertheless, an average ratio of 3 has been observed on real programs at CINECA. For the FCI code, a ratio ranging from 2 to 4 is observed; the beta-beta scaling is better on the T3E, the alpha-beta scaling is similar on the two machines.

To understand better the reported trends, we study the two routines' behavior with respect to their main components. In Figure 7 the alpha-beta behavior and its main components (`SAXPY`, `Scatter`, `Get`) are reported. The dark lines show the total routine behavior (the same as in Fig. 6), and the light lines show, from top to bottom, the values relative to `Get`, `SAXPY`, and `Scatter` parts of the routine. It is evident that the performance of the whole routine is driven by the `SAXPY` and `Scatter` parts; they are more important in terms of computational effort.

Similar considerations can be applied to the beta-beta routine, which is reported, together with its main components (`SAXPY` and `List`), in Figure 8. Here, no communication activity or stridden data movement is present. The total ratio is driven by the `SAXPY` component. Only for a high number of processors does the `List` component become comparable, slightly slowing the total ratio.

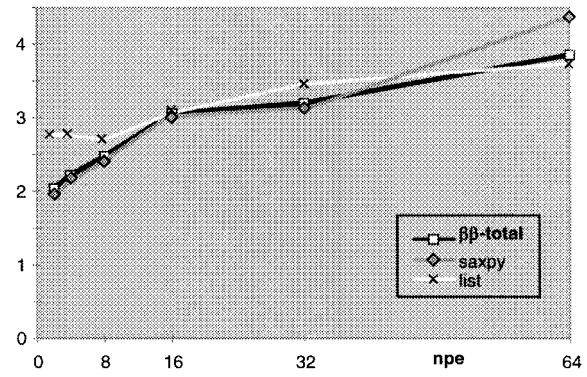

FIGURE 8. Timing comparison between Cray T3D and Cray T3E, N_2 case, beta-beta routine and its components.

In Figure 9 we report timing relative to a run on 128 processors of the acetylene system with B1 basis set. Because of its dimension, it can be run only on 128 T3D processors (reducing the work arrays), whereas, on the T3E, it can fit, without buffering work areas, on a minimum of 96 processors. Data reported in Figure 9 are consistent with those reported in the previous discussion of the N_2 case. In particular, the ratio between T3D and T3E timing is 3.6 for the alpha-beta routine and 2.7 for the beta-beta routine.

Conclusions

The implementation of a modified version of our parallel full-CI code has been described. With respect to previous versions of our algorithm, the present version is characterized by the use of a direct list algorithm to compute the lists of mono- and biexcitations each time they are needed. This required a deep modification in the data distribution of the CI vectors among the different processors, at the expense of a small performance degradation. However, this modification enabled us to treat full-CI cases of dimensions substantially larger than those possible with our previous algorithms.

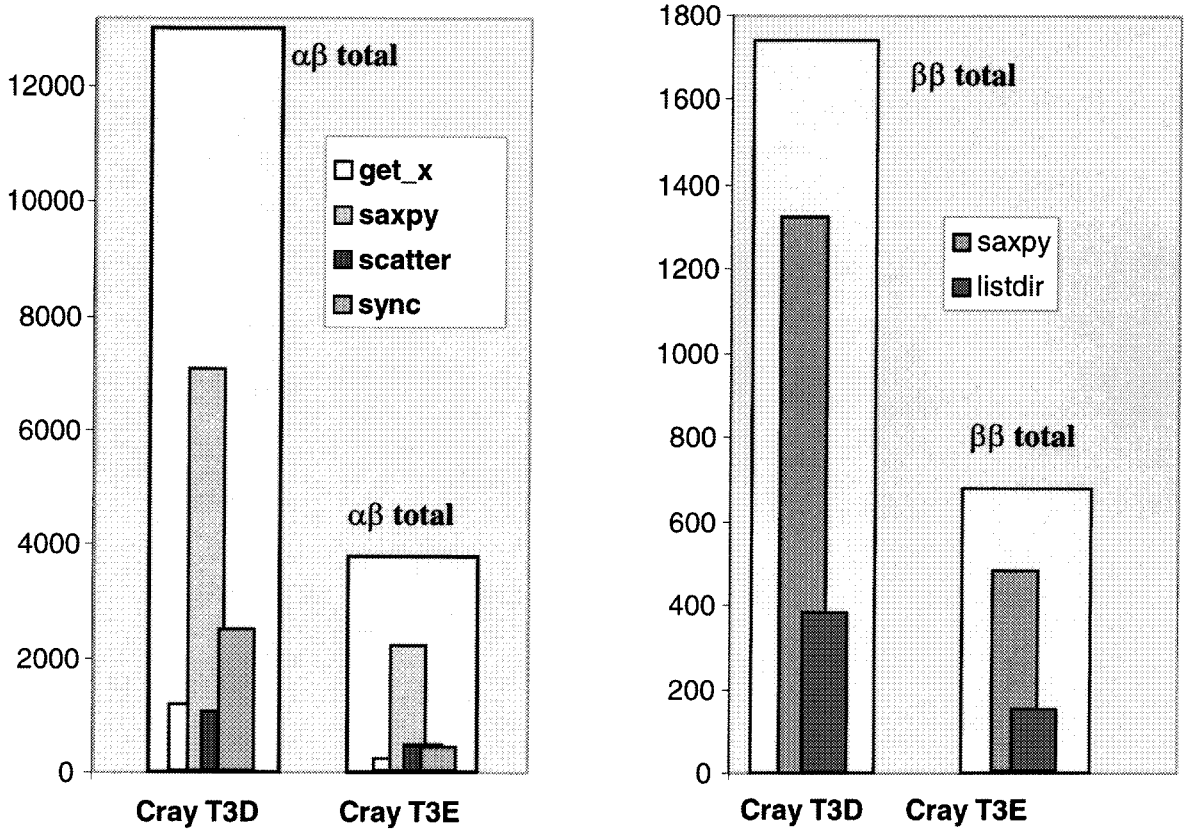

FIGURE 9. Timing comparison between Cray T3D and Cray T3E, on 30 orbitals, acetylene case, alpha-beta and beta-beta routines together with their components.

To treat the huge amount of I/O activity when studying very large CI spaces, we used a strategy consisting of preliminary computing of a very accurate starting vector (guess) for the real full-CI calculation. This guess is obtained through a full-CI calculation where the very small elements of both X and Y are discarded at the moment of storage on disk. This fact implies an important savings in the I/O time, without notably affecting the total number of iterations needed to achieve convergence.

We applied the new version of the algorithm to the acetylene molecule, with two different basis sets. By using the larger one, we were able to compute the ground state energy at a geometry close to equilibrium, with a full-CI space of more than 5 billion symmetry-adapted Slater determinants. Several different geometries are currently being investigated using the smaller bases set, with a full-CI space of about 2.5 billion Slater determinants. The different geometries correspond to different total-symmetric stretches of the molecule and will be used to assess the reliability of truncated-CI methods.

Acknowledgments

All computations were performed on CINECA Cray T3D and Cray T3E, within the ''Supercomputing Grant'' programme.

References

1. N. C. Handy, *Chem. Phys. Lett*., **74,** 280 (1980).
2. P. Saxe, H. F. Schaefer III, and N. C. Handy, *Chem. Phys. Lett*., **79,** 202 (1981).
3. P. E. M. Siegbahn, *Chem. Phys. Lett*., **109,** 417 (1984).
4. P. J. Knowles and N. C. Handy, *Chem. Phys. Lett*., **111,** 315 (1984).
5. J. Olsen, B. O. Roos, P. Joergensen, and H. J. AA. Jensen, *J. Chem. Phys*., **89,** 2185 (1988).
6. P. J. Knowles, *Chem. Phys. Lett*., **155,** 513 (1989).
7. P. J. Knowles and N. C. Handy, *J. Chem. Phys*., **91,** 2396 (1989).
8. P. J. Knowles and N. C. Handy, *Comput. Phys. Commun*., **54,** 75 (1989).
9. S. Zarrabian, C. R. Sarma, and J. Paldus, *Chem. Phys. Lett*., **155,** 183 (1989).
10. R. J. Harrison and S. Zarrabian, *Chem. Phys. Lett*., **158,** 393 (1989).
11. J. Olsen, P. Jorgensen, and J. Simons, *Chem. Phys. Lett*., **169,** 463 (1990).
12. G. L. Bendazzoli and S. Evangelisti, *J. Chem. Phys*., **98,** 3141 (1993).
13. R. Ansaloni, E. Rossi, and S. Evangelisti, *Lecture Notes Comput. Sci*., **919**, 488 (1995).
14. S. Evangelisti, G. L. Bendazzoli, R. Ansaloni, and E. Rossi, *Chem. Phys. Lett*., **232,** 353 (1995).
15. S. Evangelisti, G. L. Bendazzoli, R. Ansaloni, F. Durì, and E. Rossi, *Chem. Phys. Lett*., **252,** 437 (1996).
16. L. Gagliardi, L. Bendazzoli, and S. Evangelisti, *J. Comp. Chem*., **18,** 1329 (1997).
17. E. Rossi, R. Ansaloni, and S. Evangelisti, *Proceedings of the Fall 1994 CUG Meeting*, p. 64, K. Winget, Shepherdstown, WV, 1994.
18. R. Shepard, *J. Comput. Chem*., **11,** 382 (1992).
19. R. J. Harrison and R. Shepard, *Annu. Rev. Phys. Chem*., **45,** 623 (1994).
20. Cray T3E and Cray T3D Programming Environment Differences, SR-2199 2.0.1 (1996).
21. MPI-2: Extensions to the Message-Passing Interface, document of the MPI-forum; Web page at URL: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.
22. Cray T3E Fortran Optimization Guide, SG-2518 3.0 (1997).
23. Introducing CrayLibs IN-2167 2.0 (1995).
24. CF90 Fortran Language Reference Manual, SR-3902, SR-3903, SR-3905 2.0 (1995).